O'REILLY®

Python in Education

Teach, Learn, Program

Nicholas H. Tollervey

Python in Education

Teach, Learn, Program

Nicholas H. Tollervey

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Python in Education
by Nicholas H. Tollervey

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor: Kristen Brown
Copyeditor: Gillian McGarvey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2015: First Edition

Revision History for the First Edition
2015-03-11: First Release

978-1-491-92462-4

[LSI]

In memory of John Pinner (1944–2015), who tragically lost his fight with cancer while this work was being written.

He taught Python to me and many other people in the UK. This kind and gentle man understood the value of computing education and, through his work as founder and chairperson of the PyconUK conference, promoted it with great zeal.

He is a sorely missed mentor, leader and friend.

Table of Contents

Preface ..... **vii**

1. The Story of Python **1**

2. A Pythonic Case Study: Raspberry Pi **7**

3. Why Python in Education? **13**
Code Readability 13
Obvious Simplicity 15
Open Extensibility 20
Cross-Platform Runability 21
Humanity 24

4. Python's Secret Weapon: Community! **25**
The PSF 26
Conferences 27

5. Looking Ahead **29**

Preface

Programming is cool.

Why?

Because programmers are obviously black-clad kung-fu ninjas with "hacker aliases" who always take the red pill, type really fast to "crack databases," and save the world from renegade artificial intelligences.[1] Or perhaps programmers are geeky wunderkind who make billions of dollars by founding social networks that have more users than most countries have citizens.[2] Then again, programmers are those socially awkward yet rather useful savants who are always so keen to fix your computer (but never let them explain what they're doing or they'll bore you with overly enthusiastic technobabble).[3] Also, why is programming such a "boy" thing?

Stereotypes stop people from doing stuff.

This applies as much to programming as any other aspect of life. If your view of a programmer is as ridiculous as the stereotypes listed above, then programming is likely to appear as an intimidating form of technical magic or a dull obsession for misfit boys who avoid the great outdoors.

But there is hope: the damaging prejudices and misconceptions listed above can be overturned through education. Organizations such as the RaspberryPi Foundation and One Laptop Per Child (OLPC)

1 Neo, in the film *The Matrix*.

2 Mark Zuckerberg's character in the film *The Social Network*.

3 Moss, from the UK TV series *The I.T. Crowd*.

project see programming as a means of empowerment. A new generation of programmers are learning to be enterprising digital makers and creators rather than merely passive users. Even politicians are waking up to the realization that the long-term viability of their country's economy and public services fundamentally depends on citizens' ability to excel in the digital realm. And so educators have been tasked to change the school computing curriculum from an uninspiring Microsoft Office how-to into an education that includes programming, taking control of the computer and making it do things where the only encumbrance is one's imagination.

There is a programming language whose creator has explicitly said that his aim is to make computer programming for everybody. That person is Guido van Rossum, and the programming language is Python.

Python Is Everywhere

A quick glance online suggests that Python is the language du jour for teaching programming. Yet Python is, and has been for a while, one of the world's most popular programming languages in industry as well. Every day, without realizing it, you probably use software that is written using Python. Python is used by companies to write all sorts of applications. Google, NASA, Bank of America, Disney, CERN, YouTube, Mozilla, The Guardian—the list goes on of companies and organizations of all sizes in all sectors of the economy that use Python.

Why is Python so popular?

I aim to answer this question from an educational perspective. One might distill the answer into the following points:

Resources
: There are lots of resources for learners of all ages and levels. These range from traditional textbooks to websites that offer online self-paced courses in Python programming. With the advent of the Raspberry Pi and OLPC projects, everyone can get hold of affordable hardware that runs Python.

The Language's Design
: Python is easy to learn, intuitive, pleasing to the eye and comes with a plethora of libraries that allow programmers to build all sorts of applications addressing different domains and activities.

It's easy to read: if you squint a little, most Python code is comprehensible even to people who wouldn't call themselves programmers (for example, Python is very popular among scientists).

Community
: Python has a large, diverse and proactive community associated with it. The Python Software Foundation (PSF) is a community-led charitable organization whose mission is to promote, protect and advance the Python programming language.

Momentum
: Being popular is itself a strength and a virtuous circle that reinforces Python's popularity. New projects and initiatives are announced all the time. For example, the author is aware of several yet-to-be publicly announced Python-in-education projects. The online version of this document will be updated to reflect these announcements, so be sure to check *http://www.oreilly.com/programming/free/python-in-education.csp*.

What's in It for You?

If you're reading this report, I imagine you're a programmer, teacher, student, parent or other interested party. You're probably wondering how this report will help you understand Python's place in the recent resurgence of interest in computing education. Assuming the categories of reader listed above, here's what's in it for you.

You're a Programmer

If you already know how to write code, then you might believe that education is of little interest to you.

But wait!

If you're a *good programmer*, you also know that part of the vocation of software development involves learning new technology and, when in a position of responsibility, teaching junior colleagues how your software works. Put simply, to be a programmer is to be both a teacher and a student.

To describe programming so children understand you indicates that you know your craft at a deep level. For instance, you appreciate what to leave in or how much to leave out of an explanation. You

have clear enough mental models of the concepts of programming that you can accurately analogize and summarize. Furthermore, you explain yourself in simple and easy-to-understand language that demonstrates your own clarity of thought. Finally, finding the opportunities to practice these skills on young coders is a sign of moral and professional value: you're putting something back into the wider community and have shown initiative.

This report describes how you and your colleagues may continue your professional development by supporting the next generation of programmers.

You're a Teacher

Well done! Before becoming a programmer, I was a senior secondary school teacher in the United Kingdom. It was the most difficult yet also most rewarding job I have ever had to do. Teaching is the one profession that creates all the other professions. It is a calling (you're certainly not doing it for the money or perks) and, as a practitioner of this remarkable profession, if you're looking for help and support in teaching programming, then you've made a great choice by investigating Python.

This report describes where to learn about Python (so you're no longer one page ahead of the class), get involved with and find support from the wider Python community and become acquainted with the story of Python—an interesting subject in itself when teaching computing.

You're a Student

It often seems daunting to learn new skills and knowledge. But rest assured, Python is both relatively easy to learn and a real programming language used widely in industry. Python comes with "batteries included": there are plenty of libraries of code written in Python that allow you to build all sorts of amazing and incredible applications.

Python's community is a welcoming and friendly place. Remember, what you get out of the community is directly related to what you put in. Don't just sit there, do something! Jump in and get involved.

This report explains where to learn Python (so you're several pages ahead of your teacher in programming classes) and how to get in touch with the wider community.

You're a Parent

Someone you care about is obviously passionate about computers and programming. That's a good thing—if they make it their career, they're joining a profession that has a high demand for quality engineers.

This report gives you enough information so you can best support your loved one. Hopefully, it will allay any fears and uncertainties you may have and answer some of your questions about learning to program with Python.

You're Interested in Learning More

You've probably heard about the computing revolution in schools. Maybe you've heard of the Raspberry Pi. In any case, Python is at the center of these fundamental changes in computing education.

This report arms you with the facts and information you need to understand where Python sits within this context.

Acknowledgments

Many thanks to Amelia Watkiss, Samuel Tollervey and William Tollervey for the moment of adventuring into Python captured on the front cover of this document. Thanks also to Carrie Anne Philbin, Naomi Ceder and Tim Golden for invaluable feedback on an early draft. The picture of a fractal tree built in Minecraft was provided by the extraordinarily creative Martin O'Hanlon. Finally, Meghan Blanchette has, yet again, been a very patient editor.

CHAPTER 1
The Story of Python

In December 1989, a Dutch programmer called Guido van Rossum was looking for a "hobby" project to keep him occupied over his Christmas holiday. He decided to write an interpreter for a new programming language he'd been thinking about. He states that he was in a slightly irreverent mood so he decided to call his project "Python" after the famous British comedy troupe, "Monty Python's Flying Circus."[1]

Van Rossum goes on to explain:

> It all started with ABC, a wonderful teaching language that I had helped create in the early eighties. It was an incredibly elegant and powerful language, aimed at non-professional programmers. Despite all its elegance and power and the availability of a free implementation, ABC never became popular in the Unix/C world. I can only speculate about the reasons, but here's a likely one: the difficulty of adding new "primitive" operations to ABC. It was a monolithic, "closed system," with only the most basic I/O operations: read a string from the console, write a string to the console. I decided not to repeat this mistake in Python.

Perhaps this explains why Python is so popular in education: from the beginning, it was derived from a language designed for teaching and aimed at nonprofessional programmers. Yet by making it an open and extensible platform (Python is an open source project),

1 From the foreword to Mark Lutz's book *Programming Python* (1st ed.), published by O'Reilly. *http://www.python.org/doc/essays/foreword/*

Python could grow into the hugely popular and flexible language it is today, capable of simply and effectively addressing many different types of computational problems.

Van Rossum is now the Benevolent Dictator For Life (BDFL) for the Python language and continues to make core contributions to the language along with many thousands of developers spread all over the world. From such curious beginnings, Python has grown to be a major open source software project. Why? What is it about Python that has made it so successful? What are the guiding principles that attract such a large group of programmers, both amateur and professional, to work with and contribute to Python?

A handy answer is the Zen of Python. Its author, Tim Peters, describes it as a document that "succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down."

To read the Zen of Python, one simply starts the Python interpreter and types the command `import this`:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious
way to do it.
Although that way may not be obvious at first unless
you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

While much of this cultural artifact won't make sense to nonprogrammers, the general sense—which should be obvious to all—is a striving for simplicity, clarity, pragmatism and a sense of playful fun. Contrast this outlook with the usual stereotypes of programming languages as complex, obscure, dense and serious endeavours. It's hard not to wonder about Python, "What's not to like?"

As Alex Martelli puts it in his *Python Cookbook* (O'Reilly), "To describe something as clever is not considered a compliment in the Python culture."

Python culture? Yes, there is a Python culture that labels positive aspects of Python programming as "Pythonic." A simple, elegant and easy-to-comprehend solution to a programming problem (i.e., it conforms to the Zen of Python) will often be called Pythonic.

Python's focus on simplicity, clarity, pragmatism and fun is appealing in an engineering context. I believe it is also essential and attractive in the world of education. After all, engaging young coders with a text-based programming language puts up plenty of barriers to entry (learning to type accurately, underdeveloped literacy and comprehension skills and a lack of syntactic discipline when writing spring immediately to mind). This is before even having to deal with the complexity of the language itself, its idioms and abstractions.

Python's potential role in the world of education was not missed by Van Rossum. In 1999, he made his position on the subject public through a proposal for a project called "Computer Programming for Everybody: A Scouting Expedition for the Programmers of Tomorrow". The opening paragraphs of the proposal succinctly describe his outlook:

> In the seventies, Xerox PARC asked: "Can we have a computer on every desk?" We now know this is possible, but those computers haven't necessarily empowered their users. Today's computers are often inflexible: the average computer user can typically only change a limited set of options configurable via a "wizard" (a lofty word for a canned dialog), and is dependent on expert programmers for everything else.
>
> We ask a follow-up question: "What will happen if users can program their own computer?" We're looking forward to a future where every computer user will be able to "open the hood" of their computer and make improvements to the applications inside. We believe that this will eventually change the nature of software and software development tools fundamentally.

The project planned to have three components. They intended to:

1. Develop a new computing curriculum suitable for high school and college students.
2. Create better, easier-to-use tools for program development and analysis.
3. Build a user community around all of the above, encouraging feedback and self-help.

The results of the project's endeavors were to come together in a scientific exploration of the role of programming in the next generation of computing environments. The proposal continues:

> We intend to start with Python, a language designed for rapid development. We believe that Python makes a great first language to learn: Unlike languages designed specifically for beginners, Python is also the choice of many programming professionals. It has an active, growing user community which has already expressed much interest in this proposal, and we expect that this will be a fertile first deployment ground for the teaching materials and tools we propose to create. During the course of the research we will evaluate Python and propose improvements or alternatives.

Exploring how learners used Python was going to inform the development of new programming languages and tools. These opening paragraphs also beautifully encapsulate Python's strengths in the context of education.

Unfortunately the project was never finished due to a lack of funding. I find it an interesting (and rather frustrating) "what if?". How might Python and computing education have developed if the project had delivered on all three of the planned components?

In any case, this is yet more evidence of how Python has always had education as a core focus. The proposal also appears prescient given the recent changes in attitude to the computing curriculum and the promotion of programming. Van Rossum was a decade and a half too early. Could such a project be revived today?

Nevertheless, such educational endeavors did not go unnoticed. Projects concerned with computing and education have successfully made use of Python to great effect. For example, the One Laptop Per Child (OLPC) project has the following aim:

> We aim to provide each child with a rugged, low-cost, low-power, connected laptop. To this end, we have designed hardware, content and software for collaborative, joyful, and self-empowered learning. With access to this type of tool, children are engaged in their own education, and learn, share, and create together. They become connected to each other, to the world and to a brighter future.

The user interface and applications for the OLPC were written in Python. Over 2.5 million children and teachers in 42 countries have such laptops.

Perhaps the most famous and successful computing-in-education project in history is the Raspberry Pi (with over 5 million devices delivered so far). Unsurprisingly, Python is at the heart of the project. The next chapter is a case study exploring why the Raspberry Pi Foundation chose to focus on Python and how this has led to some unexpected yet wonderful outcomes.

CHAPTER 2

A Pythonic Case Study: Raspberry Pi

> The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.
>
> What's more, the Raspberry Pi has the ability to interact with the outside world, and has been used in a wide array of digital maker projects, from music machines and parent detectors to weather stations and tweeting birdhouses with infra-red cameras. We want to see the Raspberry Pi being used by kids all over the world to learn to program and understand how computers work.
>
> —Raspberry Pi Foundation

In a BBC interview, Eben Upton, one of the founders of the Raspberry Pi project, explained that the device was so named because it was capable of running the Python programming language. He later conceded at Pycon 2013 that their spelling might have been a bit off ("Pi" instead of "Py").

When talking with Eben and two members of the Raspberry Pi education team, Carrie Anne Philbin and Ben Nuttall, it's clear that

Python is an important aspect of their work.[1] They initially chose to concentrate on Python for several reasons.

The traditional first lesson in any programming language is to make the computer print "Hello World" on the screen. This is ridiculously easy in Python:

```
print('Hello World')
```

Contrast this with the Java version:

```
public class java {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Eben explained that it's a great illustration of Python's simplicity, readability and accessibility. Compared to the Java code, it has minimal syntax and is a meaningfully concise instruction of what you, the programmer, are expecting the computer to do. In contrast, with Java you have two options: either ignore four out of five lines of code, or learn enough to understand object-oriented programming, Java method definitions, string arrays and referencing the standard library.

Given such an easy way to start programming with Python, the Raspberry Pi Foundation valued the way the language allows learners to move on. As we know, Python is a real programming language rather than simply an educational "toy" language (such as the rather wonderful visual programming tool Scratch). Learners can graduate to real-world programming using the language they're already familiar with. Eben went on to explain, "Python has a learning curve with no discontinuity in it. Python is very smooth."

Finally, and perhaps because several of the creators of the Raspberry Pi were children in the 1980s, it is possible to use Python in a similar fashion to the old 8-bit home computers from the 1980s that ran BASIC. This is good because learners simply start to type code and the computer immediately responds, thus creating a tight feedback loop that encourages exploration and experimentation (important aspects of any learning activity).

1 In mid-February 2015, the author visited the Raspberry Pi Foundation to discuss their use of Python.

In fact, the Raspberry Pi was originally going to boot into a Python console (à la the home computers of the 1980s).

Given these initial considerations and the worldwide success of the project, it is interesting to learn about the Raspberry Pi Foundation's subsequent experience with Python. Did their initial focus on Python turn out to be a good decision?

Unsurprisingly, it did.

They described the Python community as "exactly the sort you want. Education is a core part of the community."

Visiting Pycon in 2013 bore fruit. First, the Foundation was able to observe the Python community engaging in an education track for kids. Second, it was an occasion to test the hypothesis that kids don't code because they don't have a platform to program on. This hypothesis is important given that modern desktop PCs and mobile devices don't encourage fun "hacks" (used in the positive sense: the exploration and elegant use of technology). That the kids at Pycon 2013 were engaged and having a lot of fun initially proved the hypothesis. This has subsequently been confirmed by the device's international runaway success in the education world.

Since then, Pythonic highlights for the Foundation have included contributions of code to the project. As you can see in Figure 2-1, the Raspberry Pi has general purpose input/output (GPIO) pins that allow users to attach and control external devices.

Figure 2-1. A Raspberry Pi (note the GPIO pins running across the top of the board)

Ben Croston, an enterprising tuba-playing brewer (tuba players love to drink beer) set up a microbrewery that used the Raspberry Pi to monitor and adjust the brewing process. In order to make this happen, he wrote and then released as free software a Python library for easily controlling devices via the GPIO pins. Since then, the library originally used for brewing beer for members of brass bands in northern England has been used for a huge number of physical computing projects. Put simply, whenever you hear of a Raspberry Pi project where the device is used to connect to and control something else, it probably uses Ben's library. Such sharing of Python code creates opportunities for inadvertent educational use: kids are plugging in and controlling all sorts of hardware devices (such as the weather stations and tweeting birdhouses mentioned in the description from the Raspberry Pi Foundation at the start of this chapter).

You may have heard of Minecraft, a hugely popular game that works like a sort of digital Lego. The game is set in a computer-generated "blocky" world. Players have the ability to build and explore this world along with others connected on the network. Mojang, the publisher of Minecraft, released a version for the Raspberry Pi and included a Python-based library that made it easy for anyone to interact with the game via code.

There have been some amazing projects that make use of this library, especially the work of Martin O'Hanlon and David Whale who have created many resources and projects that inspire kids to interact with their favorite game via Python. For example, at PyconUK 2014, Martin worked with about 80 kids who collaborated together to program such projects as in-game teleporters, magic bridge building (walk off a cliff and a bridge will magically appear at your feet), and growing multi-colored fractal trees, as seen in Figure 2-2.

Figure 2-2. Multi-colored fractal trees created in Minecraft using Python running on a Raspberry Pi

The image on the front cover of this report was also taken at PyconUK. Notice the lack of adult supervision; this is a group of kids between the ages of 6 and 9 autonomously working out how to program Minecraft with Python. These are tomorrow's programmers inspired into programming *today*.

Another interesting side effect of having the Raspberry Pi at PyconUK was how such educational activities motivated professional software developers. A group of Python developers took the library provided by Mojang and started to add features and update it to the latest version of Python, making the learning curve for the kids shallower and smoother.

As Ben from the Raspberry Pi education team pointed out, teachers don't necessarily have the skills or time to write the Python libraries they need to create engaging lessons for their students. Yet the Python community steps up and makes stuff happen so that their work can be repurposed (sometimes unintentionally) for use in the classroom.

Such crossover between programming and education, facilitated by the popularity of the Raspberry Pi device in both realms, has led to many positive, mutually beneficial outcomes. Teachers now make up a significant minority of attendees at PyconUK, kids attend Python conferences all over the world, Python developers run code workshops (for example, the London Python Code Dojo contrib-

uted several code projects to the Raspberry Pi foundation) and treat teachers as colleagues who generate use cases for them to turn into working code.

Ultimately, the efforts and good will shown by the Python community have paid off: the Raspberry Pi Foundation has given back to the community by funding Python-related events, providing resources and supporting programming projects such as PyPy, a high-performance version of Python that runs extraordinarily quickly on the Raspberry Pi (and other devices).

The future looks bright for the Raspberry Pi. For example, it's going to fly with British ESA astronaut Tim Peake to the International Space Station. British schoolchildren will program the device with experiments and tasks for Tim to fulfill as part of the AstroPi project.

I bet you can't guess the language in which these programs will be written.

CHAPTER 3

Why Python in Education?

I am going to answer a very simple question: which features of the Python language *itself* make it appropriate for education? This will involve learning a little Python and reading some code. But don't worry if you're not a coder! This chapter will hopefully open your eyes to how easy it is to learn Python (and thus, why it is such a popular choice as a teaching language).

Code Readability

When I write a to-do list on a piece of paper, it looks something like this:

```
Shopping
Fix broken gutter
Mow the lawn
```

This is an obvious list of items. If I wanted to break down my to-do list a bit further, I might write something like this:

```
Shopping:
    Eggs
    Bacon
    Tomatoes
Fix broken gutter:
    Borrow ladder from next door
    Find hammer and nails
    Return ladder!
Mow the lawn:
    Check lawn around pond for frogs
    Check mower fuel level
```

Intuitively we understand that the main tasks are broken down into sub-tasks that are indented underneath the main task to which they relate. This makes it easy to see, at a glance, how the tasks relate to each other.

This is called scoping.

Indenting in this manner is also how Python organizes the various tasks defined in Python programs. For example, the following code simply says that there is a function called `say_hello` that asks the user to input their name, and then—you guessed it—prints a friendly greeting:

```
def say_hello():
    name = input('What is your name? ')
    print('Hello, ' + name)
```

Here's this code in action (including my user input):

```
What is your name? Nicholas
Hello, Nicholas
```

Notice how the lines of code implementing the `say_hello` function are indented just like the to-do list. Furthermore, each instruction in the code is on its own line. The code is easy to read and understand: it is obvious which lines of code relate to each other just by looking at the way the code is indented.

Most other computer languages use syntactic symbols rather than indentation to indicate scope. For example, many languages such as Java, JavaScript and C use curly braces and semicolons for this purpose.

Why is this important?

If, like me, you have taught students with English as an additional language or who have a special educational need such as dyslexia, then you will realize that Python's intuitive indentation is something people the world over understand (no matter their linguistic background). A lack of confusing symbols such as '{', '}' and ';' scattered around the code also make it a lot easier to read Python code. Such indentation rules also guide how the code should look when you write it down—the students intuitively understand how to present their work.

Compared to most other languages, Python's syntax (how it is written) is simple and easy to understand. For example, the following

code written using the Perl programming language will look for duplicate words in a text document:

```
print "$.: doubled $_\n" while /\b(\w+)\b\s+\b\1\b/gi
```

Can you work out how Perl does this?

(In Perl's defense, it is an amazingly powerful programming language with a different set of aims and objectives than Python. That's the point—you wouldn't try to teach a person how to read with James Joyce's *Ulysses*, despite it being widely regarded as one of the top English-language novels of the 20th century.)

Put simply, because you don't have to concentrate on how to read or write Python code, you can put more effort into actually understanding it. Anything that lowers the effort required to engage in programming is a good thing in an educational context (actually, one could argue that this is true in all contexts).

Obvious Simplicity

The simple core concepts and knowledge required to write Python code will get you quite far. That they are easy to learn, use and remember is another characteristic in Python's favor. Furthermore, Python is an obvious programming language—it tries to do the expected thing and will complain if you, the programmer, attempt to do something clearly wrong. It's also obvious in a second sense—it names various concepts using commonly understood English words.

Consider the following two examples.

In some languages, if you want to create a list of things, you have to use variously named constructs such as arrays, arraylists, vectors and collections. In Python, you use something called a list. Here's my to-do list from earlier written in Python:

```
todo_list = ['Shopping', 'Fix broken gutter', 'Mow the lawn']
```

This code assigns a list of values (strings of characters containing words that describe tasks in my to-do list) to an object named `todo_list` (which I can reuse later to refer to this specific list of items).

In some languages, if you want to create a data dictionary that allows you to store and look up named values (a basic key/value

store), you'd use constructs called hashtables, associative arrays, maps or tables. In Python, you use something called a dictionary. Here's a data dictionary of a small selection of random capital cities:

```
capital_cities = {
    'China': 'Beijing'
    'Germany': 'Berlin',
    'Greece': 'Athens',
    'Russia': 'Moscow',
    'United Kingdom': 'London',
}
```

I've simply assigned the dictionary to the `capital_cities` object. If I want to look up a capital city for a certain country, I reference the country's name in square brackets next to the object named `capital_cities`:

```
capital_cities['China']
'Beijing'
```

Many programming languages have data structures that work like Python's lists and dictionaries; some of them do the obvious thing and call such constructs "lists" and "dictionaries"; some other languages make using such constructs as easy and obvious as Python (although many don't). Python's advantage is that *it does all three of these things*: it has useful data structures as a core part of the language, it gives them obvious names, and makes them extraordinarily easy to use. Such usefulness, simplicity and clarity is another case of removing barriers to engaging with programming.

As mentioned earlier, Python also does the expected thing. For example, if I try to sum together an empty dictionary and an empty list (something that's obviously wrong—evidence that I've misunderstood what I'm trying to do) Python will complain:

```
>>> {} + []
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'list'
```

This is simply telling me that I can't use the "+" operand to sum a dictionary and a list. This is to be expected, rather obvious and quite helpful.

Nevertheless, other languages try to be less strict and more forgiving of the programmer. While this may sound like a good idea, it means that faulty code like that attempted above will be executed without any error and cause uncertain results (after all, what is the answer of

summing a dictionary and a list?). Here's what the ubiquitous JavaScript language will do when you attempt to add the equivalent data structures (in JavaScript parlance, an object '{}' and an array '[]'):

```
> {} + []
0
```

Of course, the answer is obviously zero!?!

Guess what happens if you try to sum an empty array with an object in JavaScript (we switch around the summed terms).

```
> [] + {}
"[object Object]"
```

I bet you were expecting a consistent result!

(Again, the caveat of JavaScript having a different set of aims and objectives to Python should be applied here.)

Learning, by its very nature, involves making mistakes and realizing that mistakes have been made. Only then can behavior be adjusted and progress made. If you're learning to program using a language like JavaScript (that would rather make what appears to be a best guess at what you mean, rather than complain about an obvious error), then all sorts of mistakes will pass by unnoticed. Instead, you'll either continue in your mistaken view of the programming world or you'll have to understand the rather complex and tortuous rules that JavaScript uses to cause `{} + []` to equal `0` and `[] + {}` to equal `"[object Object]"` (itself, a difficult educational feat to pull off).

Python's simplicity and obviousness encourages learners and professional developers alike to create understandable code. Understandable code is easier to maintain and less likely to contain bugs (because many bugs are caused by *misunderstanding what the code is actually doing* compared to what you *mistakenly think it ought to be doing*). Being able to simply state your ideas in code is a very powerful and empowering capability.

For example, consider an old-school text adventure game. Players wander around a world consisting of locations that have descriptions and exits to other locations. The program below very clearly and simply implements exactly that.

Most of the program consists of either comments to explain how it works or is a data dictionary that describes the game world. It is only

the final block of code that actually defines the behavior of the game. It is my hunch, even if you're not a programmer, that you'll be able to get the gist of how it works.

```
"""
A very simple adventure game written in Python 3.

The "world" is a data structure that describes the game
world we want to explore. It's made up of key/value fields
that describe locations. Each location has a description
and one or more exits to other locations. Such records are
implemented as dictionaries.

The code at the very end creates a game "loop" that causes
multiple turns to take place in the game. Each turn displays
the user's location, available exits, asks the user where
to go next and then responds appropriately to the user's
input.
"""


world = {
    'cave': {
        'description': 'You are in a mysterious cave.',
        'exits': {
            'up': 'courtyard',
        },
    },
    'tower': {
        'description': 'You are at the top of a tall tower.',
        'exits': {
            'down': 'gatehouse',
        },
    },
    'courtyard': {
        'description': 'You are in the castle courtyard.',
        'exits': {
            'south': 'gatehouse',
            'down': 'cave'
        },
    },
    'gatehouse': {
        'description': 'You are at the gates of a castle.',
        'exits': {
            'south': 'forest',
            'up': 'tower',
            'north': 'courtyard',
        },
    },
    'forest': {
        'description': 'You are in a forest glade.',
```

```
        'exits': {
            'north': 'gatehouse',
        },
    },
}


# Set a default starting point.
place = 'cave'
# Start the game "loop" that will keep making new turns.
while True:
    # Get the current location from the world.
    location = world[place]
    # Print the location's description and exits.
    print(location['description'])
    print('Exits:')
    print(', '.join(location['exits'].keys()))
    # Get user input.
    direction = input('Where now? ').strip().lower()
    # Parse the user input...
    if direction == 'quit':
        print('Bye!')
        break  # Break out of the game loop and end.
    elif direction in location['exits']:
        # Set new place in world.
        place = location['exits'][direction]
    else:
        # That exit doesn't exist!
        print("I don't understand!")
```

A typical "game" (including user input) looks something like this:

```
$ python adventure.py
You are in a mysterious cave.
Exits:
up
Where now? up
You are in the castle courtyard.
Exits:
south, down
Where now? south
You are at the gates of a castle.
Exits:
south, north, up
Where now? hello
I don't understand!
You are at the gates of a castle.
Exits:
south, north, up
Where now? quit
Bye!
```

Furthermore, from an educational point of view, this simple adventure game can be modified in all sorts of interesting and obvious ways by learners: adding objects to the world, creating puzzles, adding more advanced commands and so on. In fact, there are opportunities for cross-curricular work with other disciplines. Playing such a game is a form of interactive fiction—perhaps the English department could help the students come up with more than just the bare-bones descriptions of the original?

Open Extensibility

Despite the powerful simplicity of the core language, programmers often need to reuse existing library modules of code to achieve a common task. A library module is like a recipe book of instructions for carrying out certain related tasks. It means programmers don't have to start from scratch or reinvent the wheel every time they encounter a common problem.

While most programming languages have mechanisms to write and reuse libraries of code, Python is especially blessed in having a large and extensive standard library (built into the core language), as well as a thriving ecosystem of third-party modules.

For example, a common task is to retrieve data from a website. We can use the `requests` third-party module to download web pages using Python:

```
>>> import requests
>>> response = requests.get('http://python.org/')
>>> response.ok
True
>>> response.text[:42]
'<!doctype html>\n<!--[if lt IE 7]>   <html '
```

(This code tells Python that we want to use the requests library, gets the HTML for Python's home page, checks that the response was a success [it was] and displays the first 42 characters of the resulting HTML document.)

Some modules, such as `requests`, do one thing and do it exceptionally well. Other modules are organized into large libraries to create application frameworks that solve many of the repetitive tasks needed when writing common types of application.

For example, Django is an application framework for writing web applications (as used by Mozilla, The Guardian, National Geographic, NASA and Instagram, among others). Django looks after common tasks such as modelling data, interacting with a database, writing templates for web pages, security, scalability, deciding where to put business logic and so on. Because this has already been taken care of by Django, developers are able to concentrate on the important task of designing websites and implementing business logic.

Many languages have extensive code libraries and application frameworks, but Python's strength is in its broad reach. SciPy and NumPy are used by scientists and mathematicians, NLTK (the Natural Language Tool Kit) is used by linguists parsing text, Pandas is used extensively by statisticians and OpenStack is used to organize and control cloud-based computing resources. The list goes on and on.

Teaching a language that has such extensive real-world use has an obvious benefit: learners acquire a skill in a programming language that has real economic value.

Another aspect of being openly extensible is that Python is an open source project. Anyone can take part in developing the language by submitting bug reports or patches to the core developers (led by Guido van Rossum himself). There is also a well-understood and simple process by which new features of the language are proposed and implemented: the Python Enhancement Proposals (PEPs). In this way, the language is developed in full view of the community that uses it with the opportunity for the community to inform and guide its future development. This process is explained in great detail by PEP 1.

Cross-Platform Runability

Python is a platform-agnostic language: it works on Microsoft Windows, Mac OS X, Linux and many other operating systems and devices. It's even possible to have Python as a service through websites such as Python Anywhere.

This is important in an educational context because Python works on the computers used in schools. Students can also use it on the computers they have at home, no matter the make or model they may own. Furthermore, Python as a service provided via a website is an excellent solution to the problem of the infamously troll-like

school system administrators who won't let teachers install anything other than Microsoft Office on school PCs. Users simply point their browser at a website and they are presented with a fully functional Python development environment without having to install any additional software.

As we have seen, Python also works on small-form devices such as the Raspberry Pi. It even runs on microcontrollers—small, low-powered chips designed to run within embedded devices, appliances, remote controls and toys (among other things).

The MicroPython project has created a pared-down version of Python 3 optimized for such devices, and provides a small electronic circuit board (see Figure 3-1) that runs such a svelte version of Python.

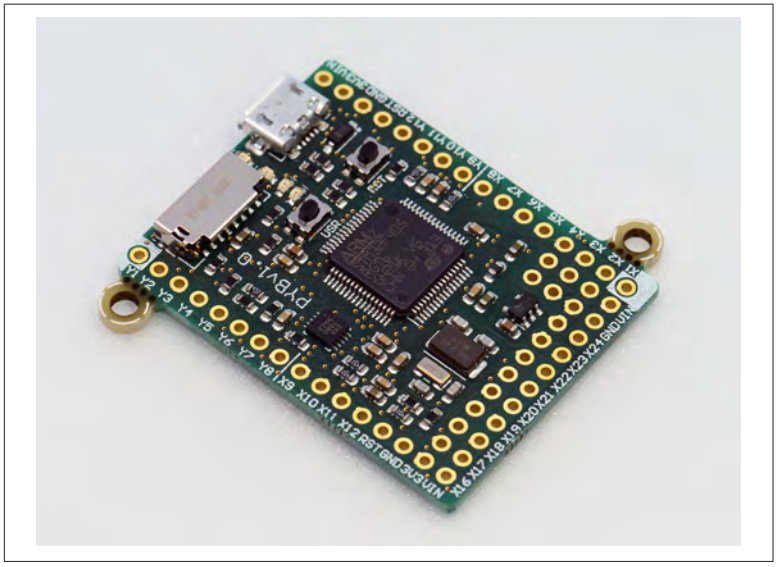

Figure 3-1. A MicroPython board (about the same size as a postage stamp)

This very simple Python-based operating system can be used as the basis for all sorts of interesting and fun electronic projects. Add-ons for the board include an LCD display screen, speaker and microphone and motors. It is a relatively easy task to build a simple robot with such a device.

More recently, the BBC announced the MicroBit project, a small battery-powered programmable device for children. One million of the devices will be given away to 11-year-olds in the UK at the start of the new academic year in September 2015. It will also be on sale to the public.

The MicroBit fastens to clothing and has a couple of buttons and an LED matrix that displays scrolling text and images. Just like the Raspberry Pi and Micro Python board, it has the ability to interact with other devices via I/O connections.

Python is one of three supported languages for programming the MicroBit.

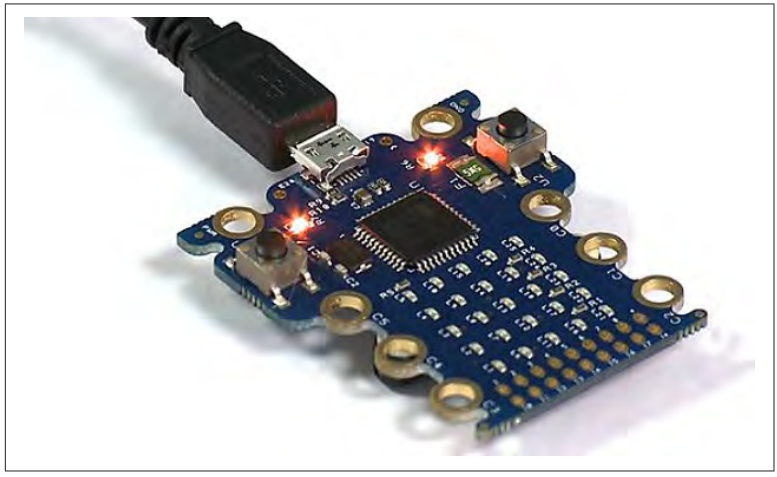

Figure 3-2. A prototype of the BBC's MicroBit programmable device for children

The important educational advantage is continuity.

By learning Python, a student has access to all sorts of fun and interesting platforms that can be explored using tools and code they're familiar with. Because Python runs on so many platforms, it is feasible to write code for one device and, assuming it doesn't use device-specific code and run within hardware constraints, it should run on many others.

Humanity

While not a strict feature of the language, Python's community, history and philosophy often shines through code written in Python.

The odd Monty Python reference (the website that hosts third-party Python modules is called the Cheese Shop after the sketch about a cheese shop with no cheese), a playful sense of fun (for instance, "the PyPy project" is so named because it is a high-performance version of Python written in Python) and other apparent eccentricities bestow upon Python the appearance of an approachable and interesting language. It's obviously used by humans for humans rather than being an abstract tool for esoterics.

Python's community is a friendly and diverse bunch. It is to this community of developers, teachers and students that I want to turn in the next chapter.

Put simply, the Python community is the secret weapon of its success.

CHAPTER 4

Python's Secret Weapon: Community!

Why is Python's community so important?

When you learn a new skill, you become aware of and a participant in the unique culture associated with that thing. The military, teachers, musicians and other vocations all have their own characteristic and immediately recognizable cultures. The same goes for programmers and different programming languages.

Happily, the Python community has an excellent reputation for being a friendly group of people who value openness, actively engage in outreach and give up their time for educational support. These are all attributes that make it easy for both teachers and students to get involved with Python's inimitable culture. As Eben Upton from the Raspberry Pi Foundation mentioned, the Python community is "exactly the sort you want. Education is a core part of the community."

Python programmers (variously called Pythonists, Pythonistas and/or Pythonauts) are also a well organized bunch and have created the Python Software Foundation (PSF) as a rallying point for the community. It also means that there is a legal entity with which governments, companies and other institutions can formally interact.

The PSF

Here's how the PSF describes itself:

> The Python Software Foundation (PSF) is a volunteer led organization devoted to advancing open source technology related to the Python programming language. It qualifies under the US Internal Revenue Code as a tax-exempt 501(c)(3) scientific and educational public charity, and conducts its business according to the rules for such organizations.

The PSF was created to promote, protect, and advance the Python programming language and to support and facilitate the growth of a diverse and international community of Python programmers. This is achieved by supporting the development of the Python programming language itself (whose intellectual property belongs to the PSF), providing technical infrastructure for the Python community (such as servers, mailing lists and the Python website), running and supporting various international Python conferences (or Pycons,[1] such as the one held in the UK mentioned in Chapter 2), and the giving of grants to individuals and organizations for projects related to the development of Python, Python-related technology, and educational resources.

Anyone who is a user or supporter of Python can join and volunteer as little or much as they see fit. The Python website and PSF should be your first port of call for information relating to the Python community. It includes the complete documentation for the language (and tutorials, too).

The PSF also hosts several mailing lists that cater to various locales and interests. For example, there is an education special interest group mailing list that you can join (the web page for the Edu-SIG also includes many useful links for resources and evidence of Python's efficacy as an educational programming language).

Another important aspect of the PSF's work is outreach and helping to make the community a welcoming place for newcomers—no matter their background, age or level of experience. This is manifested in several ways.

1 For a complete list of Pycons around the world, see *http://pycon.org/*.

Conferences supported by the PSF must have a code of conduct that helps to promote and maintain the community's reputation as a friendly, welcoming and dynamic group of people. Put simply, they help to make it clear that conference attendees are expected to treat each other in a way that reflects the widely held view that diversity and friendliness are strengths of the community to be celebrated and fostered.

The PSF awards grants for projects that promote Python, Python-related technology, educational programs and resources. This is an important mechanism for community-led support and development —if you have an idea for something to contribute that needs funding, you should apply (the process is easy and the board are responsive and helpful).

Conferences

Like every international community of free software developers, many members collaborate over the Internet rather than in real life. As a result, conferences are an important part of the community because they literally bring people together. Friendships are strengthened, collaborators are found and ideas are debated. Code is furiously written during "code sprints" (intense days of group programming). There are also the usual conference events: talks, tutorials, dinners and keynote speeches.

More recently in the world of Python conferences, things have taken a decidedly educational turn. Since 2012, PyconUK has had teachers attend and give presentations. In 2013, PyconUK had a specialist education track for teachers and developers to come together and learn from each other. Since 2013, the education track at PyconUK has had a day set aside for kids to attend with their families. Next year, PyconUK expects about 50 teachers and 150 kids to attend over the course of two days during the main conference.

In North America, there has been a PyCon Education Summit for developers and teachers since 2013 (and since as early as 2003, there have been education-themed "open spaces"). Also in 2013, a kids' track was initiated where developers volunteer their time to teach young coders and help them take their first steps into the world of Python.

Pycon Australia will hold their first education "miniconf" in 2015 and evidence from discussions at Europython 2014 suggests that many European countries are in the early stages of making their own national conferences education-friendly with tracks for teachers and/or students.

Such educational efforts are not limited to conference tracks for teachers and kids. Underrepresented groups in the wider technology sector have had their educational needs met by the Python community: PyLadies is an international network of chapters providing mentorship and support for women who want to take a more active role in the Python community, Django Girls organizes free Python and Django workshops for women, and Trans*Code runs hack days that draw attention to transgender issues while focusing on introductory programming courses for those not currently working in technology.

The Python community is active, engaged and enthusiastic.

Why not get involved?

CHAPTER 5
Looking Ahead

In 20 years time, I hope to attend Pycon 2035. If not Pycon 2035, then I want to attend the conference for whatever Python and its community morphs into. I'm looking forward to working with the kids of today who are just starting out on their journey as programmers. I'll be at the end of my career but I'm certain I'll be surprised, energized and inspired by what they do.

How can I be so certain?

I'm already surprised, energized and inspired by what the kids of today do when they attend the PyconUK education track that I help to organize.

This latent talent, joie de vivre and receptiveness to programming in Python has already been identified by the Python community who want to support, cherish and foster it. I believe the renewed focus by politicians and teachers on computing will find keen allies in the Python world.

The fruits of the work done by today's Python community will ensure that there is a legacy of new generations of programmers who are empowered to be enterprising and autonomous digital makers and creators rather than mere passive users.

The future will be increasingly influenced and controlled by computing. We need to ensure that tomorrow's citizens are equipped with the skills to flourish in this world.

Python and its community can help with that.

About the Author

Nicholas Tollervey is a classically trained musician, philosophy graduate, teacher, writer and software developer. He's just like this biography: concise, honest and full of useful information.